Algorithmica (2004) 40: 133–145
DOI: 10.1007/s00453-004-1100-7

Algorithmica

New Results in Minimum-Comparison Sorting

Marcin Peczarski[1]

Abstract. We prove that sorting 13, 14 and 22 elements requires 34, 38 and 71 comparisons, respectively. This solves a long-standing problem posed by Knuth in his famous book *The Art of Computer Programming*, *Volume* 3, *Sorting and Searching*. The results are due to an efficient implementation of an algorithm for counting linear extensions of a given partial order. We also present some useful heuristics which allow us to decrease the running time of the implementation.

Key Words. Optimal sorting, The Ford–Johnson algorithm, Counting linear extensions.

1. Introduction. The problem of finding optimal sorting algorithms is one of the fundamental and fascinating problems in the theory of sorting. The problem of finding minimum-comparison sorting algorithms is especially interesting. Steinhaus posed this problem in the second edition of *Mathematical Snapshots*. Knuth devoted a part of his famous book *The Art of Computer Programming*, *Volume* 3, *Sorting and Searching* to this problem.

Let $S(N)$ be the minimum number of comparisons sufficient to sort N elements. The *information-theoretic lower bound* tells us that

$$S(N) \geq \lceil \lg N! \rceil = c_N.$$

Ford, Jr. and Johnson [2] discovered an algorithm, called *merge insertion*, which nearly and sometimes even exactly matches the theoretical lower bound. Let F_N be the number of comparisons required to sort N elements by merge insertion. We have

$N =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$c_N =$	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41	45	49	53	57	62	66	70
$F_N =$	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42	46	50	54	58	62	66	71

One can see that $S(N) = c_N = F_N$ for $N \leq 11$ and for $N = 20, 21$. After the discovery of merge insertion the first unknown value was $S(12)$. Wells discovered [11], carrying an exhaustive computer search, that $S(12) = 30$. Knuth posed the problem of finding the next value $S(13)$ in his classic book [4, Chapter 5.3.1, Exercise 35]. He conjectured that $S(13) = 33$ and $S(14) = 37$ [6]. We show that $S(13) = 34$, $S(14) = 38$ and $S(22) = 71$. The Ford–Johnson algorithm turns out to be optimal for 13, 14 and 22 elements. Our results are based on a smart implementation of the algorithm of Wells. The proof method is a further development of the method presented in [7].

[1] Institute of Informatics, Warsaw University, ul. Banacha 2, 02-097 Warszawa, Poland. marpe@mimuw.edu.pl.

Received February 5, 2003; revised March 6, 2004. Communicated by R. Sedgewick.
Online publication May 24, 2004.

2. Preliminaries. Let $U = \{u_0, u_1, \ldots, u_{N-1}\}$ be an N-element set to be sorted. Sorting U can be viewed as a sequence of partially ordered sets (U, r), where r is a partial order over U. If U is fixed we will identify the poset (U, r) with r. The sequence starts from the total disorder $r_0 = \{(u, u) : u \in U\}$ and ends with a linearly ordered set $\{u_{i_0} < \cdots < u_{i_{N-1}}\}$. Every subsequent poset is obtained from the previous one as the result of a comparison between two elements of U. Let r be a poset after performing $c - 1$ comparisons and suppose that elements u_j and u_k are being compared in the next step. Without loss of generality we can assume that $(u_j, u_k) \notin r$ and $(u_k, u_j) \notin r$. If the answer to the comparison is *element u_j is less than element u_k* then the transitive closure of the set $r \cup \{(u_j, u_k)\}$ is the new poset. We denote it by $r + u_j u_k$.

DEFINITION 1. If r and l are posets such that $r \subseteq l$ and l is a linear order then l is called a *linear extension* of r.

Every poset has at least one linear extension. Let $e(r)$ denote the number of linear extensions of a poset r. In particular every totally unordered n-element set has $n!$ linear extensions and every linearly ordered set has exactly one linear extension. Every poset r can be sorted using c comparisons only if c comparisons are sufficient to obtain a linear order from r. The following lemma generalizes the information-theoretic lower bound to all posets.

LEMMA 1. *If a poset r can be sorted using c comparisons then $e(r) \leq 2^c$.*

For a poset r the poset $r^* = \{(u, v) : (v, u) \in r\}$ is called the *dual poset* to r.

DEFINITION 2. Two posets r_1 and r_2 are *congruent* if r_1 and r_2 or r_1 and r_2^* are isomorphic.

It is easy to see that:

LEMMA 2. *Congruent posets need the same number of comparisons to be sorted.*

Let (U, r) be a poset and let A be a subset of U. The poset $(A, r \cap A \times A)$ is denoted by $r|A$. If $v \in U$ then $r[v]$ denotes the set of elements which are larger than v, precisely $r[v] = \{u : (v, u) \in r \wedge u \neq v\}$. Then we have $r^*[v] = \{u : (u, v) \in r \wedge u \neq v\}$.

3. The Method of Wells. In this section we describe briefly the method which was used to discover that there is no 29-step sorting procedure for 12 elements [11].

The set S_c contains posets which are obtained after c comparisons. The function $search(S_c, r)$ returns TRUE iff the set S_c contains a poset congruent to the poset r.

$S_0 := \{r_0\}$, where $r_0 = \{(u_0, u_0), (u_1, u_1), \ldots, (u_{N-1}, u_{N-1})\}$
for $c := 1$ **to** c_N **do**
  $S_c := \emptyset$
  for each $r \in S_{c-1}$ **do**
    for $j := 0$ **to** $N - 2$ **do**
      for $k := j + 1$ **to** $N - 1$ **do**
        if $(u_j, u_k) \notin r$ **and** $(u_k, u_j) \notin r$ **then**
          $r_1 := r + u_j u_k$
          $r_2 := r + u_k u_j$
          if $search(S_c, r_1) =$ FALSE **and** $search(S_c, r_2) =$ FALSE **then**
            $t_1 := e(r_1)$
            $t_2 := e(r_2)$
            if $t_1 \le 2^{c_N - c}$ **and** $t_2 \le 2^{c_N - c}$ **then**
              if $t_1 \ge t_2$ **then**
                $S_c := S_c \cup \{r_1\}$
              else
                $S_c := S_c \cup \{r_2\}$

We begin from the set S_0 containing only one, totally unordered, poset r_0. In step c every poset $r \in S_{c-1}$ is examined for every unrelated pair u_j and u_k in r in order to verify whether it can be sorted in the remaining $c_N - c + 1$ comparisons. As the result of the comparison of u_j and u_k one can get one of two posets $r_1 = r + u_j u_k$ or $r_2 = r + u_k u_j$. If the number of linear extensions of r_1 (r_2) exceeds $2^{c_N - c}$ then r_1 (r_2) cannot be sorted in the remaining $c_N - c$ comparisons (by Lemma 1). It follows that in this case, in order to finish sorting in $c_N - c + 1$ comparisons, elements u_j and u_k should not be compared in step c. If the numbers of linear extensions of both r_1 and r_2 do not exceed $2^{c_N - c}$ then we store one of them for a further analysis. In principle we can choose any of them. It does not influence the correctness of the method. Since the poset with the larger number of linear extensions seems to be harder to sort we keep precisely this one. Ties can be broken arbitrarily. Observe that if S_c contains a poset congruent to one of the posets r_1 or r_2, then we do not need to keep any of them (by Lemma 2). This reduces substantially the number of posets to be considered.

If the final set S_{c_N} does not contain a linearly ordered set then we conclude that sorting N elements requires more than c_N comparisons. On the other hand, if a linearly ordered set belongs to the set S_{c_N} it does not mean that it is always possible to sort N elements using c_N comparisons. This happens in the cases $N = 13$ and $N = 14$ where S_{c_N} contains a linear order and the method fails.

4. The Function Sortable. In this section we propose an improvement of the method described in the previous section. The presented modification enables us to overcome the case where the set S_{c_N} contains a linear order. To begin we prove a simple lemma.

LEMMA 3. *If N elements can be sorted using c_N comparisons then for every c, $0 \le c \le c_N$, the set S_c contains at least one poset which can be sorted using $c_N - c$ comparisons.*

PROOF. Induction on c. The lemma is obviously true for the initial set S_0. Let r be a poset in S_{c-1} which can be sorted in $c_N - c + 1$ comparisons. It follows that there exists a sorting procedure which sorts r in $c_N - c + 1$ steps. Suppose that the procedure applying to r compares elements u_j and u_k in step c. Hence $r_1 = r + u_j u_k$ and $r_2 = r + u_k u_j$ can be sorted using $c_N - c$ comparisons. By Lemma 1 $e(r_1) \le 2^{c_N - c}$, $e(r_2) \le 2^{c_N - c}$ and S_c contains a poset congruent to r_1 or r_2. Hence S_c contains a poset sortable in $c_N - c$ comparisons by Lemma 2. □

In what follows we extensively use the following corollary.

COROLLARY 1. *If for some c the set S_c does not contain a poset sortable in $c_N - c$ comparisons then there is no sorting procedure for N elements using c_N comparisons.*

We consider the recursive function *sortable* described below. The function verifies whether a given poset r can be sorted using c comparisons.

```
boolean function sortable(r, c)
  if e(r) ≤ 6 then
    return TRUE
  for j := 0 to N − 2 do
    for k := j + 1 to N − 1 do
      if (u_j, u_k) ∉ r and (u_k, u_j) ∉ r then
        r_1 := r + u_j u_k
        r_2 := r + u_k u_j
        if e(r_1) ≤ 2^{c−1} and e(r_2) ≤ 2^{c−1} then
          if sortable(r_1, c − 1) and sortable(r_2, c − 1) then
            return TRUE
  return FALSE
```

We keep the following invariant: *the parameters r and c satisfy $e(r) \le 2^c$*. The invariant ensures that the recursion always stops because c decreases in each subsequent call and the number of linear extensions is a positive integer.

The theorem of Kahn and Saks [3] states that if poset r is not a linear order then there exists a pair of unrelated elements u_j, u_k such that $1 \le e(r_1)/e(r_2) \le \frac{8}{3}$, where $r_1 = r + u_j u_k$ and $r_2 = r + u_k u_j$. It follows that if $e(r) \le \min\{6, 2^c\}$ then poset r is sortable in c comparisons. If $e(r) = 1$ then r is a linear order and we do not need any comparison to sort it. If $e(r) = 2$ then $c \ge 1$ and we can choose a pair which split r into r_1, r_2 such that $e(r_1) = e(r_2) = 1$. If $e(r) = 3$ then $c \ge 2$ and we can split r into r_1, r_2 such that $e(r_1) = 2$, $e(r_2) = 1$. If $e(r) = 4$ then $c \ge 2$ and we can split r into r_1, r_2 such that $e(r_1) = e(r_2) = 2$. If $e(r) = 5$ then $c \ge 3$ and we can split r into r_1, r_2 such that $e(r_1) = 3$, $e(r_2) = 2$. If $e(r) = 6$ then $c \ge 3$ and we can split r into r_1, r_2 such that $e(r_1) = e(r_2) = 3$ or $e(r_1) = 4$, $e(r_2) = 2$. Since $c - 1$ comparisons suffice to sort r_1 and r_2 then c comparisons suffice to sort r. Therefore the function *sortable* returns TRUE for $e(r) \le 6$. This inequality cannot be improved considerably. Figure 3 shows the poset (4th in the 2nd row) which has eight linear extensions and cannot be sorted

in three comparisons. There is an open question if every poset which has seven linear extensions can be sorted in three comparisons.

If the number of linear extensions of r is larger than six we consider posets $r_1 = r+u_ju_k$ and $r_2 = r+u_ku_j$ for each pair of unrelated elements u_j and u_k in r. If the number of linear extensions of r_1 or r_2 exceeds 2^{c-1} then by Lemma 1 comparing u_j and u_k does not lead to sorting r in c comparisons. In the latter case we verify recursively whether r_1 and r_2 are sortable in $c-1$ comparisons. Observe that the invariant is maintained. If both r_1 and r_2 are sortable then r is also sortable and the function returns TRUE. If there is no pair of sortable posets r_1 and r_2 then the function returns FALSE.

In order to use Corollary 1 we determine for every set S_c its subset S_c^* which contains only posets sortable in $c_N - c$ comparisons. Unfortunately we cannot call the function *sortable* for all posets because of its exponential complexity in c. Instead of that we consider the function *sortable*2 which use information about posets in the set S_{c+1}^* and sparingly call the function *sortable*.

```
S*_{c_N} := S_{c_N}
for c := c_N − 1 to 0 step −1 do
    S*_c := ∅
    for each r ∈ S_c do
        if sortable2(r, c_N − c, S*_{c+1}) then
            S*_c := S*_c ∪ {r}

boolean function sortable2(r, c, S*)
    for j := 0 to N − 2 do
        for k := j + 1 to N − 1 do
            if (u_j, u_k) ∉ r and (u_k, u_j) ∉ r then
                r_1 := r + u_j u_k
                r_2 := r + u_k u_j
                f_1 := search(S*, r_1)
                f_2 := search(S*, r_2)
                if f_1 and f_2 then
                    return TRUE
                else if f_1 and e(r_2) ≤ 2^{c−1} then
                    if sortable(r_2, c − 1) then
                        return TRUE
                else if f_2 and e(r_1) ≤ 2^{c−1} then
                    if sortable(r_1, c − 1) then
                        return TRUE
    return FALSE
```

If $r \in S_c$ then $2^{c_N-c-1} < e(r) \le 2^{c_N-c}$. Hence if the set S_{c_N} is not empty then it contains only one linear order and $S_{c_N}^* = S_{c_N}$ always. (It is not hard to see that $S_{c_N-1}^* = S_{c_N-1}$ and $S_{c_N-2}^* = S_{c_N-2}$ too.) For $c < c_N$ poset $r \in S_c$ is sortable in $c_N - c$ comparisons iff there exists a pair of unrelated elements u_j, u_k such that poset $r_1 = r + u_ju_k$ or poset $r_2 = r + u_ku_j$ belongs to the set S_{c+1} (we identify congruent posets) and both posets are sortable in $c_N - c - 1$ comparisons. Therefore poset r belongs

to the set S_c^* iff $r_1 \in S_{c+1}^*$ and r_2 is sortable in $c_N - c - 1$ comparisons or $r_2 \in S_{c+1}^*$ and r_1 is sortable in $c_N - c - 1$ comparisons.

5. Useful Heuristics. Counting linear extensions is the most time-consuming operation in the method described above. In order to reduce the number of counting operations we apply a few heuristics.

Recursive calls in the function *sortable* should be performed in the lazy manner. We can suppose that among r_1 and r_2 the poset with the larger number of linear extensions is harder to sort. Therefore a possibility that only one call will be sufficient is larger when the function sortable is called first for the poset with the larger number of linear extensions. However, if the first call returns TRUE then the second call is still necessary. One can show a poset r and elements u_j and u_k such that $e(r_1) > e(r_2)$, r_1 is sortable but r_2 is not sortable (using the same number of comparisons).

For a poset (U, r) let $V = \{v \in U$: there exists $u \in U, u \neq v, (u, v) \in r$ or $(v, u) \in r\}$. We have:

- for $u, v \in U \backslash V$ and $w \in V$ the posets $r + uw$, $r + vw$ are congruent and the posets $r + wu$, $r + wv$ are congruent as well;
- for $u, v, x, y \in U \backslash V$ and $u \neq v$, $x \neq y$ the posets $r + uv$, $r + xy$ are congruent.

Therefore if $V = \{u_0, u_1, \ldots, u_{n-1}\}$ it is sufficient to compare only the following pairs of elements:

- u_j and u_k, for $0 \leq j \leq n-2$ and $j+1 \leq k \leq n-1$;
- u_j and u_n, for $0 \leq j < n < N$;
- u_n and u_{n+1}, for $n < N - 1$.

Furthermore, we can start the method of Wells from the second step. All posets obtained after the first comparison are congruent and set S_1 always contains only one poset. We can assume that this is the poset $r_0 + u_1 u_0$.

It is known that $e(r_1) + e(r_2) = e(r)$ [4], [11]. The number $e(r)$ is computed in the step before. Hence one can store $e(r)$ and compute only one value: $e(r_1)$ or $e(r_2)$. In practice one of them can be calculated faster. Because counting of linear extensions can take time proportional to their number (see Section 6) then intuitively the smaller value should be computed faster. We can assume that the "better ordered" poset has the smaller number of linear extensions. If $|r^*[u_j]| + |r[u_k]| \geq |r[u_j]| + |r^*[u_k]|$ then r_1 is "not worse ordered" than r_2 and in this case we compute

$$t_1 = e(r_1), \qquad t_2 = e(r) - t_1.$$

We compute

$$t_2 = e(r_2), \qquad t_1 = e(r) - t_2$$

in the other case. Furthermore, we already know that if $V = \{u_0, u_1, \ldots, u_{n-1}\}$ and $n < N-1$ then $r + u_{n+1}u_n$ and $r + u_n u_{n+1}$ are congruent. Hence we have $e(r + u_{n+1}u_n) = e(r + u_n u_{n+1}) = e(r)/2$. In the function *sortable*2 we can check conditions $e(r_2) \leq 2^{c-1}$ and $e(r_1) \leq 2^{c-1}$ quickly using differences of known values: $e(r_2) = e(r) - e(r_1)$ and $e(r_1) = e(r) - e(r_2)$, respectively.

The next improvement follows the lemma:

LEMMA 4. *If* $(x, u) \in r$, $(v, y) \in r$ *and* $(v, u) \notin r$ *then* $e(r + xy) \geq e(r + uv)$.

PROOF. Observe that $r + uv$ and $(r + xy) + uv$ are identical. Since a comparison cannot increase the number of linear extensions we have

$$e(r + uv) = e((r + xy) + uv) \leq e(r + xy). \qquad \square$$

It follows that if $e(r + u_j u_k) > 2^c$ for some u_j and u_k then $e(r + xy) > 2^c$ for every x and y such that $(x, u_j) \in r$ and $(u_k, y) \in r$. Donald Knuth (personal communication) has observed that a slightly stronger result holds:

LEMMA 5. *If* $(x, u) \in r$, $(v, y) \in r$, $(v, u) \notin r$ *and* $r + uv$ *is unsortable then so is* $r + xy$.

PROOF. Let $E(r)$ denote the set of linear extensions of poset r. Then we have $E(r + uv) \subseteq E(r + xy)$. $\square$

6. Counting Linear Extensions. The problem of counting linear extensions of a given poset is #P-complete [1]. This indicates that counting linear extensions is probably not easier than generating them. Therefore we cannot expect a polynomial-time algorithm for counting, but nevertheless we show in the next section that the method described here is the fastest existing.

The method is based on the theorem given in Chapter 7.2.2 of [11]. In order to state the theorem precisely we need some new notation.

DEFINITION 3. Let (U, r) be a poset, let $D \subseteq U$ and let $d \in D$. The pair (A, B) is called an *admissible partition of* D *with respect to element* d when the following conditions are satisfied:

- $A \cup B = D \backslash \{d\}$ and $A \cap B = \emptyset$;
- if $(a, d) \in r$ and $a \neq d$ then $a \in A$;
- if $(d, b) \in r$ and $b \neq d$ then $b \in B$;
- $(b, a) \notin r$ for all $a \in A$ and $b \in B$.

THEOREM 1. *Let* (U, r) *be a poset.*

1. *If* $A, B \subseteq U$, $A \cap B = \emptyset$ *and* $(a, b) \notin r$, $(b, a) \notin r$ *for* $a \in A$ *and* $b \in B$, *then*

$$e(r|A \cup B) = e(r|A) \cdot e(r|B) \cdot \binom{|A| + |B|}{|A|}. \tag{1}$$

2. *If* $D \subseteq U$ *and* $d \in D$ *then*

$$e(r|D) = \sum_{A,B} e(r|A) \cdot e(r|B), \tag{2}$$

where the sum is taken over all admissible partitions of D *with respect to* d.

The main idea of the algorithm is to divide the problem into "simpler" subproblems and to solve them recursively. First, a given poset is partitioned into connected components (in the graph sense). Next, the number of all linear extensions is calculated independently for each component and the results are combined using (1). In order to compute the number of linear extensions of a connected component D we apply item 2 of Theorem 1. We take d such that the number of admissible partitions of D is the smallest possible. It turns out that for such d the number of related elements in $r|D$ is the largest possible. We get a set of pairs of simpler subproblems in this way. Each such subproblem is solved in the recursive manner and the results are combined using (2).

In order to improve performance of the above algorithm we stop recursion as early as possible. To this aim we do not partition small components D such that $|D| \leq 5$. If $|D| \leq 2$ then $r|D$ is a linear order (because it is connected) and $e(r|D) = 1$. If D consists of three, four or five elements we store the number of linear extensions in three auxiliary tables. In order to get the number of linear extensions of $r|D$ we compute the index $\sigma(r|D)$ in the appropriate table. If $|D| = 3$ then

$$\sigma(r|D) = \sum_{v \in D} |r[v] \cap D|.$$

If $|D| = 4$ then

$$\sigma(r|D) = \sum_{v \in D} \theta_4(|r[v] \cap D|),$$

where the function θ_4 is defined as follows:

$$\begin{aligned} i &= 0\ 1\ 2\ 3, \\ \theta_4(i) &= 0\ 1\ 4\ 10. \end{aligned}$$

For $|D| = 5$ we have

$$\sigma(r|D) = \sum_{v \in D} \theta_5(|r[v] \cap D|, |r^*[v] \cap D|),$$

where the function θ_5 is defined as follows:

$$\begin{aligned} i &= 0\ \ 1\ \ 2\ \ 3\ \ 4, \\ \theta_5(0, i) &= \ \ 0\ \ 1\ \ 3\ \ 10, \\ \theta_5(1, i) &= 0\ \ 6\ \ 8\ \ 5, \\ \theta_5(2, i) &= 1\ \ 8\ \ 1, \\ \theta_5(3, i) &= 3\ \ 5, \\ \theta_5(4, i) &= 10. \end{aligned}$$

We have that if $|D_1| = |D_2|$ and $\sigma(r|D_1) = \sigma(r|D_2)$ then $e(r|D_1) = e(r|D_2)$. Figures 1–3 illustrate this property.

Moreover, we can improve our algorithm, when we stop recursion before partition into connected components for posets over set containing less than five elements. Figure 1 shows that for posets over a three-element set we can unequivocally find out the number of linear extensions from the value of the index. Figure 2 shows that for posets over a four-element set there are some unconnected posets which cannot be distinguish by such a defined index. Some additional tests are needed. In fact we do this by counting the number of single elements when $\sigma(r) \in \{2, 5\}$.

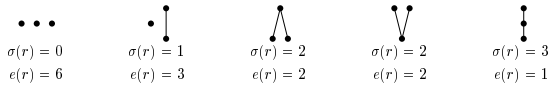

Fig. 1. All non-isomorphic posets over a three-element set.

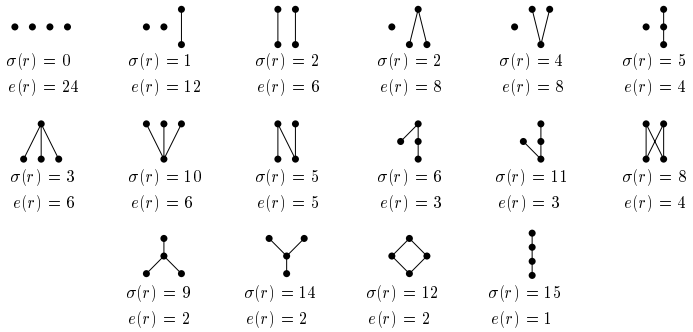

Fig. 2. All non-isomorphic posets over a four-element set.

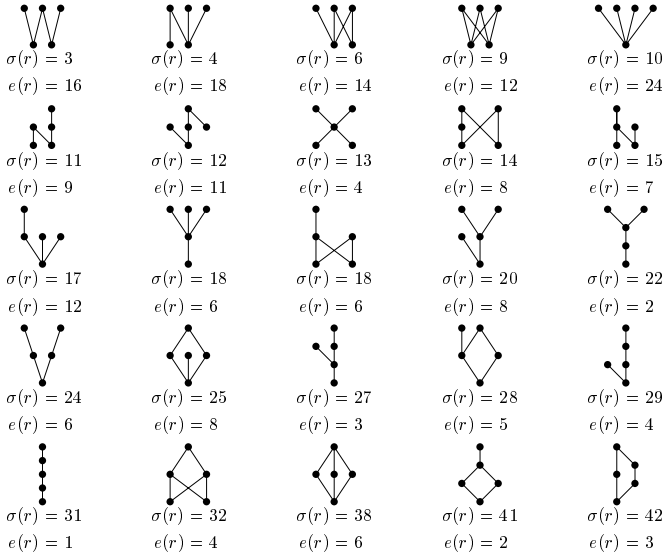

Fig. 3. All non-congruent posets over a five-element set.

Table 1. Counting linear extensions.

n	$e(r)$	This paper	Varol, Rotem	Pruesse, Ruskey
12	2,702,765	0.0006	0.2	0.07
13	22,368,256	0.0014	2.2	0.46
14	199,360,981	0.0033	17	3.4
15	1,903,757,312	0.0073	180	28
16	19,391,512,145	0.018		
21	4,951,498,053,124,096	1.1		
22	69,348,874,393,137,901	2.8		

7. Results of Experiments

7.1. *Counting Linear Extensions*. There are other algorithms which can be used for computing the exact number of linear extensions of a given poset. The improved Varol–Rotem algorithm recently published by Knuth [5] and the Pruesse–Ruskey algorithm [9] both run in time $O(e(r))$. The above algorithms are designed to generate all linear extensions of a given poset. Pruesse and Ruskey also present a modification that spares the computation time if one wants to compute only the number of linear extensions. The asymptotic time complexity of the algorithm presented in the previous section is still not known.

Table 1 compares implementations of all three algorithms. The poset under consideration was an n-element fence. Figure 4 shows an example. The computation time is given in seconds. All algorithms are implemented in the C language and compiled using the same compiler with the same optimization options. The source code of the Pruesse–Ruskey algorithm was written by Kenny Wong and Frank Ruskey and is accessible via the Internet [12]. Tests were run on a PC with a 233 MHz processor.

7.2. *Sorting* 13 *Elements*. As was already mentioned, for $N = 13$ Wells's method produces the sequence of sets $S_1, S_2, \ldots, S_{33}$ and the set S_{33} contains a linear order. However the set S_{15}^* is empty. By Corollary 1 we have that there is no procedure sorting 13 elements and using 33 comparisons. Since the Ford–Johnson algorithm sorts 13 elements using 34 comparisons we finally get that $S(13) = 34$.

Table 2 contains detailed results. Generation of the sets $S_1, S_2, \ldots, S_{33}$ and the sets $S_{33}^*, S_{32}^*, \ldots, S_{15}^*$ took 38 and 3 minutes, respectively. The experiment was run on a PC with a 650 MHz processor.

On pp. 188–191 of [4] Knuth discusses the method which allows us to consider only connected posets. This improves Wells's method for 12 elements. Table 2 shows that for 13 elements most of the considered posets are connected. Therefore we do not expect such improvement in this case.

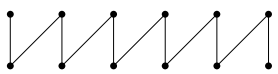

Fig. 4. 12-element fence.

Table 2. Sorting 13 elements using 33 comparisons.

c	$\lvert S_c\rvert$	Connected	$\lvert S_c^*\rvert$	c	$\lvert S_c\rvert$	Connected	$\lvert S_c^*\rvert$
1	1	1		18	391,277	386,305	11
2	2	1		19	418,412	415,543	25
3	4	2		20	374,911	373,624	42
4	8	4		21	276,858	276,400	67
5	18	8		22	171,595	171,419	105
6	31	10		23	91,010	90,961	172
7	63	26		24	41,735	41,712	292
8	171	88		25	17,129	17,125	405
9	464	267		26	6,532	6,532	482
10	1,261	752		27	2,344	2,344	439
11	3,432	2,140		28	836	836	316
12	9,087	6,276		29	266	266	169
13	22,777	17,743		30	81	81	76
14	53,308	45,844		31	30	30	30
15	110,533	101,496	0	32	6	6	6
16	199,145	190,139	1	33	1	1	1
17	304,542	297,187	6				

7.3. *Sorting* 14 *Elements*. After solution of the problem for 13 elements a question naturally appears whether 37 comparisons suffice to sort 14 elements. The strategy for 13 elements should work for 14 elements too. Initial experiments have shown that more than 50 million posets are expected in the set S_{21}. Instead we use a strategy which requires less space but more time.

First we generate sets $S_1, S_2, \ldots, S_{16}$. Table 3 shows the results. Next we split the set S_{16} into subsets containing 1000 or 2000 posets. The last subset contains 965 posets. We analyse each subset independently and obtain that the set S_{16}^* contains one poset. Using this information we determine that the set S_{15}^* is empty. Again by Corollary 1 we have $S(14) > 37$. Since $F_{14} = 38$ we get $S(14) = 38$.

Generation of the sets $S_1, S_2, \ldots, S_{16}$, the set S_{16}^* and the set S_{15}^* took 79 minutes, 390 hours and 18 minutes, respectively. The experiment was run on a PC with a 650 MHz processor.

7.4. *Sorting* 22 *Elements*. As suggested on p. 191 of [4], the case $S(22)$ is similar to the case $S(12)$. In experiments performed for 22 elements and 70 comparisons we obtain

Table 3. Sorting 14 elements using 37 comparisons.

c	$\lvert S_c\rvert$	Connected	$\lvert S_c^*\rvert$	c	$\lvert S_c\rvert$	Connected	$\lvert S_c^*\rvert$
1	1	1		9	958	554	
2	2	1		10	2,969	1,773	
3	4	2		11	9,293	5,742	
4	9	5		12	29,254	18,991	
5	21	10		13	92,005	65,055	
6	44	18		14	283,036	220,771	
7	108	51		15	825,950	701,144	0
8	318	173		16	2,194,965	1,983,215	1

Table 4. Sorting 22 elements using 70 comparisons.

c	$\lvert S_c \rvert$	Connected	c	$\lvert S_c \rvert$	Connected
1	1	1	21	29,042	13,398
2	1	0	22	46,372	24,591
3	2	1	23	66,148	39,809
4	2	0	24	83,730	56,341
5	2	0	25	92,138	67,777
6	4	0	26	87,516	69,434
7	8	4	27	73,181	61,572
8	14	6	28	54,110	47,470
9	21	5	29	35,399	32,001
10	37	8	30	20,976	19,409
11	64	15	31	11,188	10,567
12	107	31	32	5,371	5,145
13	166	32	33	2,393	2,317
14	280	48	34	872	851
15	547	146	35	320	314
16	1,202	516	36	111	111
17	2,472	1,039	37	31	31
18	4,769	1,814	38	7	7
19	9,023	3,195	39	1	1
20	16,591	6,565	40	0	0

that Wells's method stops after 39 comparisons, the set S_{40} is empty and a linear order cannot be produced. This implies that $S(22) > 70$. Since $F_{22} = 71$ we get $S(22) = 71$.

Table 4 contains detailed results. Two computers were involved in the experiment. The experiment needed 1973 hours on a 450 MHz processor and 1740 hours on a 650 MHz processor.

8. Some Aspects of Implementation. To ensure the correctness of our implementation we checked whether the program can determine properly the known values of $S(N)$ for $N \leq 12$. Assuming we can use at most $S(N) - 1$ comparisons we obtained that there is always $c \leq S(N) - 1$ for which the set S_c is empty. This confirms that $S(N) - 1$ comparisons is not enough to sort N elements. Table 5 shows the results for 12 elements.

Table 5. Sorting 12 elements using 29 comparisons.

c	$\lvert S_c \rvert$	Connected	c	$\lvert S_c \rvert$	Connected
1	1	1	13	340	280
2	1	0	14	316	275
3	2	1	15	258	246
4	3	1	16	165	160
5	4	1	17	95	92
6	8	2	18	40	39
7	15	6	19	25	25
8	36	21	20	17	17
9	71	35	21	11	11
10	116	52	22	3	3
11	183	98	23	0	0
12	260	184			

Table 6. Sorting 11 elements using 26 comparisons.

c	$\lvert S_c\rvert$	Connected	$\lvert S_c^*\rvert$	c	$\lvert S_c\rvert$	Connected	$\lvert S_c^*\rvert$
1	1	1	1	14	54,082	53,195	641
2	2	1	2	15	62,637	62,167	925
3	5	3	3	16	55,612	55,434	1,267
4	11	6	4	17	38,279	38,227	1,531
5	26	13	6	18	20,545	20,530	1,601
6	58	25	9	19	8,921	8,917	1,357
7	147	74	16	20	3,232	3,231	941
8	409	227	37	21	1,044	1,044	512
9	1,176	747	73	22	301	301	225
10	3,369	2,513	114	23	92	92	88
11	8,732	7,368	184	24	28	28	28
12	19,490	17,916	287	25	5	5	5
13	36,350	34,985	433	26	1	1	1

For $N \leq 11$ we checked whether $S(N)$ comparisons are sufficient. We obtained that each set in the sequence $S_1, \ldots, S_{S(N)}$ contains at least one sortable poset, which is consistent with Lemma 3. Table 6 shows the results for 11 elements.

A poset is represented in the tables $r[0..N-1]$ and $r^*[0..N-1]$ where $r[j]$ and $r^*[j]$ are the sets of indices of all elements larger and smaller than u_j, respectively. One poset occupies $8(N+1)$ bytes for $N \leq 12$ and $8(N+2)$ bytes for $N \geq 13$. Poset's congruence is checked using hash functions and Ullman's graph isomorphism algorithm [10]. The complete source code can be downloaded from [8].

Acknowledgments. I thank Krzysztof Diks for his comments, suggestions and encouragement and Donald Knuth for his comments on my conference paper [7].

References

[1] G. Brightwell and P. Winkler, Counting Linear Extensions, *Order*, **8** (1991), 225–242.
[2] L. Ford and S. Johnson, A Tournament Problem, *American Mathematical Monthly*, **66** (1959), 387–389.
[3] J. Kahn and M. Saks, Balancing Poset Extensions, *Order*, **1** (1984), 113–126.
[4] D. E. Knuth, *The Art of Computer Programming*, Volume 3, 2nd edn., Addison-Wesley, Reading, MA, 1998.
[5] D. E. Knuth, Generating All Permutations, in *The Art of Computer Programming*, Addison-Wesley, Reading, MA, 2002, Pre-Fascicle 2B.
[6] D. E. Knuth and E. B. Kaehler, An Experiment in Optimal Sorting, *Information Processing Letters*, **1** (1972), 173–176.
[7] M. Peczarski, Sorting 13 Elements Requires 34 Comparisons, in *Proceedings of the* 10*th Annual European Symposium on Algorithms* (R. Möhring and R. Raman, eds.), Lecture Notes in Computer Science, vol 2461, Springer-Verlag, Berlin, 2002, pp. 785–794.
[8] M. Peczarski, Home Page, http://www.mimuw.edu.pl/˜marpe.
[9] G. Pruesse and F. Ruskey, Generating Linear Extensions Fast, *SIAM Journal on Computing*, **23** (1994), 373–386.
[10] J. R. Ullman, An Algorithm for Subgraph Isomorphism, *Journal of the ACM*, **23** (1976), 31–42.
[11] M. Wells, *Elements of Combinatorial Computing*, Pergamon Press, Oxford, 1971.
[12] The Combinatorial Object Server, http://www.theory.csc.uvic.ca/˜cos.